数理解析研究所講究録 1426 巻 2005 年 39-44

配列を扱う非線形先頭再帰プログラムからの再帰除去

On Recursion Removal from Non-Linear Top-Recursive Programs

高須洋平† 酒井正彦‡ 西田直樹‡
Yohei Takasu† Masahiko Sakai‡ Naoki Nishida‡
草刈圭一朗‡ 坂部俊樹‡
Keiichirou Kusakari‡ Toshiki Sakabe‡

† 名古屋大学大学院 工学研究科
Graduate School of Engineering, Nagoya University
‡ 名古屋大学大学院 情報科学研究科
Graduate School of Information Science, Nagoya University

† takasu@sakabe.i.is.nagoya-u.ac.jp
‡ {sakai, nishida, kusakari, sakabe}@is.nagoya-u.ac.jp

概要

再帰プログラムは書きやすく読みやすいが，実行時には関数呼出しとスタック操作が必要となる．本研究では配列型のデータを操作する関数を対象とし．非線形再帰にも適用できる再帰除去法を与える．この方法は，与えられたデータの定数倍の作業領域を用いることで，スタックを用いずに動作する反復型プログラムへの変換を行なう手法である．特に関数の先頭で再帰が行なわれる型のプログラムに注目し，マージソートを例として反復型プログラムに書き換える事で高速化が可能であるかを評価する．

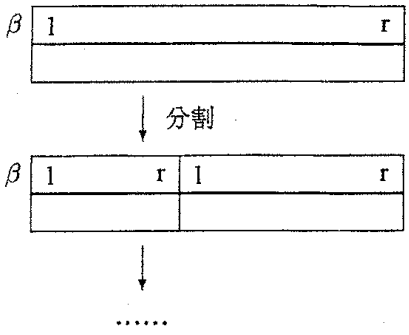
1 はじめに

与えられたプログラムをより効率的なプログラムに変換するプログラム変換について，現在に至るまで多くの研究がなされてきた [1]．その手法の一つとして，再帰プログラムを反復型プログラムに変換する再帰除去の研究が 1970 年代中頃から行われている [2][3][4]．

再帰プログラムは反復型プログラムと比較して読みやすく書きやすいが，実行時に再帰呼出しとスタック操作が必要となる．そのため，インライン展開ができない，局所参照性が悪いなどのプログラム最適化上の問題を引き起こし，さらにメモリ使用量も増加する．それ故，再帰プログラムをスタックを使用せず，計算のコストを増加させずに反復型プログラムに変換する事ができれば，メモリ使用量，及び実行時間の削減が期待できる．実際に，関数の末尾に一度だけ再帰呼び出しが行われる**線形末尾再帰関数** (一般には，単に末尾再帰関数と呼ばれる) に対しては，再帰除去が自動化され，多くのコンパイラで実装されている [5]．しかしながら，同時に複数の再帰呼出しが行われる非線形再帰については，一般的な再帰除去法が見付かっておらず，そのような変換が困難であることが知られている．

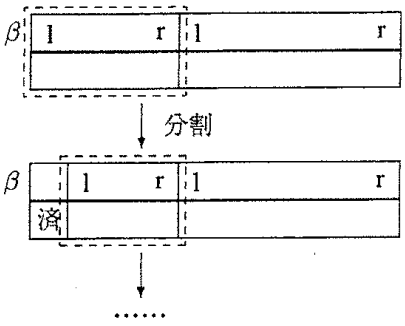
本研究では数列のソーティング，二分探索といった，配列型のデータを操作する再帰関数を対象とした再帰除去を図る．その中でも特に，マージソートのように関数の先頭で再帰呼出しを行なう関数に限定し，非線形再帰にも適用できる再帰除去手法を提案する．

再帰除去では計算のコストを増加させず，新たにメモリ領域を使用しない変換が望まれるが，本研究の手法で変換されたプログラムは配列領域を余分に用いる．この配列は，操作対象である配列と同じ，または定数倍のサイズの配列領域である．



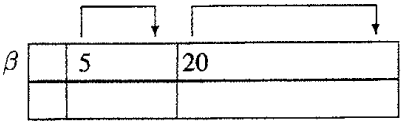
※ l: left, r: right.
操作領域を配列 β 中の印で示している.

図 1: 配列による領域の指定



※ 左端の領域から処理を行なうことで，処理の完了した領域が左端に溜る.
→ 左端に検索しなくてもよい領域ができる.

図 2: 左端からの処理



※ 領域の先頭に配列のサイズや終端へのポインタを書き込むことで，先頭の要素を参照するだけで領域の範囲を知ることができる.

図 3: ポインタの使用

本手法では最初に配列を用意し，またこれを操作する処理が必要となるが，これらの処理に要する時間が再帰呼出しによる遅延より少なければ，結果として実行時間は短縮できると期待される．この考えに基づき，マージソート関数に対して再帰除去を適用し，その評価を行なう.

2 配列を用いた再帰除去

分割ソートや二分探索といった配列を操作対象とする関数を考える．これらの関数では，引数として操作する配列に加え，配列中の処理する領域を，その先頭，末尾の値 (left, right) で受け取る．この領域に対して個々の処理及び領域の分割を行ない，分割した領域の先頭，末尾の値は新たな left, right として，再帰呼出しの際に呼び出された関数に渡される.

これに対し，提案する反復型のプログラムでは，分割領域の情報を予め用意しておいた配列 (β とする) に格納する．分割された領域を β に書き込み，これを順に読みとり処理していくことで，再帰やスタックを用いない単純な反復型プログラムとして実行できるようになる (図 1).

この際，問題となるのが配列 β から操作範囲を読み取るのに必要な計算のコストである．β の要素を一つずつ順に調べた場合，最悪で一回の反復につき $O(n)$ のコストが増加する事になる．そのため，左端の領域から順に実行する (図 2)，領域の先頭に終端へのポインタを記入する (図 3)，等により実行時間の増加を抑える.

再帰呼出しの後では再帰呼出し以外の処理を行なわない再帰の形式を，**末尾再帰**と呼称する．末尾再帰型の関数については，受け取った領域に対して，配列の操作を行なった後にそれを分割すればよい．従って，次の手順を繰り返すことにより，操作部分のコードを変更することなく再帰型の関数と同じ動作を行なうことができる.

1. 操作対象領域を配列 β から取得.
2. 操作を適用する.
3. 領域を分割，配列 β に書き込む.

この方針に基づいて，手動で再帰除去を適用した結果，非線形，かつ末尾再帰であるクイックソート関数に関しては，計算機環境により (僅かではあるが) 実行時間の減少が見られることを確認した [6].

本研究ではマージソートを例に，先頭型の再帰関数に対しての適用を考える.

3 先頭再帰関数からの再帰除去

本研究では，再帰呼出しを行う前に，操作領域の分割を除く処理を行なわない形式の再帰を**先頭**

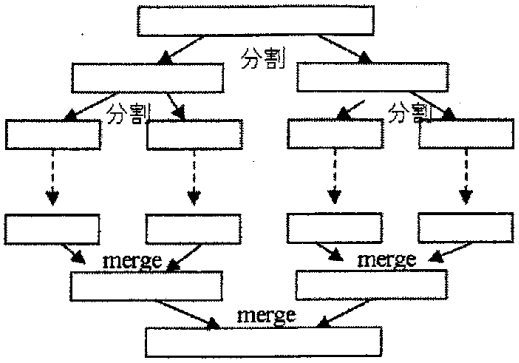


図 4: マージソート

再帰と定義する．マージソートを例とする先頭再帰プログラムでは，再帰呼出しがその他の処理より前に行なわれる．そのため，最初に全ての領域の分割のみが行なわれ，分割された領域を逆戻りに統合しながら操作を行なうことになる (図 4)．

これを模倣するため，反復型のプログラムでは処理を分割，統合の二段階に分けて行なう．分割は再帰呼出しによる領域の分割を，統合は操作を行なった後に呼出し元の関数へ返る過程を反復処理で記述するものとなる．

このとき，領域を統合する際に，統合すべき領域群の選び方が問題となる．マージソートであれば，隣接する二つの領域がマージされ，一つの領域となることが知られている．しかしながら，例えば 1 番目と 2 番目の領域がマージされるべきか，2 番目と 3 番目の領域がマージされるべきかは分からない．どちらが同じ分割元から分けられた領域であるかは，分割の手順によるからである．この選び方が間違っていると，マージソートでさえも，操作部分のコードを変更しなければ動作しなくなることがある．

再帰型のマージソートでは，再帰により形成されるスタックフレームの構造により，スタックに積まれている順に統合を行うことで，必ず分割の過程を逆順に辿ることができる．一方，再帰を用いずにマージソートを行なう場合，分割元が同じ領域のペアを別の形で知らなければならない．これを決定するため，分割の回数，すなわち再帰呼出しの深さ (*depth*) を示す値を，領域の範囲 (size) と同時に配列 β に記録する (図 5)．これにより配列 β の要素は自然数の二項組 (*depth*, *size*) となり，そのサイズは操作対象配列の二倍になる．

分割の段階では，全ての領域をサイズが 1 になるまで二分割する．分割する度に対応する領域の

図 5: 配列の用意

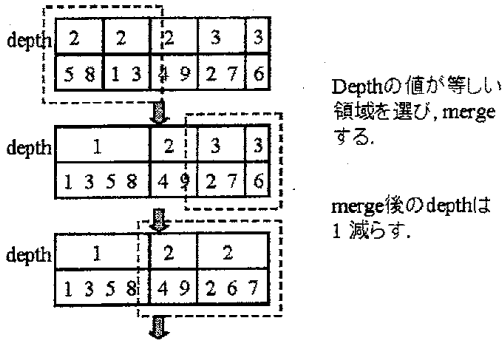


図 6: 領域の統合

depth の値を一つ増やし，これにより分割の深さを記録する．

領域の統合の際は，隣接する *depth* の値が等しい領域を統合すればよい (図 6)．このとき，統合した領域の *depth* の値を一つ減らす．これは関数の処理が終了し，再帰の呼出し元へ処理が返ることに対応する．

まとめると，反復型の先頭再帰プログラムは，以下の分割，統合の処理をそれぞれ繰り返すことで実装される．

- 分割
  1. 領域を二分割し，配列 β の *size* に書き込む．
  2. 配列 β の分割した領域の場所で，*depth* の値を増やす．
- 統合
  1. 隣接する *depth* の値が等しい領域を統合し，操作を適用する．
  2. 統合した領域を配列 β の *size* に書き込む．
  3. *depth* の値を一つ減らす．

この手順によって実装された反復型のマージソート関数を付録 A に示す．

表 1: マージソート関数の実行結果 (単位: n sec)

要素数	最適化なし -O0		
	再帰型	反復型	改善率
10,000	581	726	-25.0%
100,000	7,360	9,025	-22.5%
1,000,000	91,773	110,732	-20.7%
要素数	最適化あり -O2		
	再帰型	反復型	改善率
10,000	522	575	-10.2%
100,000	6,670	7,224	-8.3%
1,000,000	84,000	90,494	-7.7%
要素数	最適化あり -O3		
	再帰型	反復型	改善率
10,000	512	603	-17.6%
100,000	6,483	7,706	-18.9%
1,000,000	83,193	96,079	-15.5%

4 評価

コンパイラは gcc-2.95.3 を使用．反復型のマージソート関数を実装し，最適化なし (gcc の -O0 オプション) でコンパイルを行なった場合，および，最適化 (gcc の -O2 または -O3 オプション) を適用した場合のそれぞれについて，実行時間を調べた．

表 1 では，再帰型のマージソートに対して，反復型のマージソートは -O2 最適化による実行時間の減少率が多いことが分かる．これは，再帰による最適化上の問題が取り除かれたため，反復型のマージソートでは再帰型には適用できなかった最適化が行なわれているためだと考えられる．しかしながら，最適化レベルを -O2 から-O3 に上げると，反復型のマージソートは逆に効率が悪化する結果となった．

どの最適化を適用した場合であっても，期待した実行時間の改善は見られず，元プログラムである再帰型のマージソートより効率が下がってしまっている．これは範囲取得の際の配列操作による遅延が再帰呼出しによるコストを上回ったためである．遅延の主な原因として考えられるものを以下に列挙する．

depth の書き込みによる遅延: 統合するべき領域を識別するために，本研究では *depth* の値を計算し，これを読みとることで再帰の深さを表現した．再帰型の関数ではこのような値の操作は必要なく，反復型ではそのための手間が増えている．

領域の統合による遅延: 反復型のマージソートでは，二つの領域を統合してから，それに対してマージの操作を適用している．これに対し，再帰型のマージソート関数では，呼出し元の関数のパラメータが残されているため，改めて分割前の領域を計算する必要がない．つまり，反復型では各マージの段階で“二つの領域を組み合わせるための計算”が余分に必要となる．

前者は統合するべき領域を発見するためのコストであり，後者は二つの領域を統合するためのコストである．

これらの遅延を取り除くためには，分割前の領域を計算を行なわずに得る必要がある．つまり，再帰型と同様にスタックを用いて操作領域を記憶しなければならない．スタックによりパラメータを記憶することは本質的に再帰プログラムと同じであるため，本研究では使用しないことを前提としている．よって，この計算のコストの増加は避けられない．

これらの遅延をいかに少ない量に抑えられるかが変換後のプログラムの効率を決定するが，現在では実用的な値は得られていない．

5 まとめ

本研究では，固定サイズの配列領域を追加することでマージソート関数からの再帰除去を試みた．ここでは再帰型のマージソートの処理を模倣し，再帰呼出し，及び，再帰元に処理が戻る際の，パラメータの授受に関する動作のみを配列を用いて代用している．これはマージソートのみではなく，それに似た型の先頭再帰プログラムにも共通して適用できる変換手法を得ることを目的としているからである．

また，実際にマージソートの処理を反復型で記述し，実装した．しかし，実行速度の向上という点では期待される値は得られなかった．これは *depth* の操作によって，元の再帰プログラムにない処理が必要となり，計算のコストが増加してしまったことが原因だと考えられる．

depth の値は，分割元が同じ領域を識別するためのパラメータである．より簡潔な計算で分割元を知ることができれば，現状より実行時間を減らすことができる．しかし，元のマージソートを上回る効率を得られるか否かについては不明である．

6 今後の課題

提案した再帰除去手法は，特定の条件を持つプログラムのみに適用できる．前述のように，対象プログラムは先頭再帰で，かつ配列を操作対象とするものでなければならない．また，引数は配列内の領域を示す値であるとし，各々の領域は重ならずにそれぞれ独立していなければならない．本手法では一つの配列 β に全ての操作領域を記述するため，重複した領域を許すと，配列 β 内の同一要素に複数の値を書き込まなければならない状況が生まれてくるからである．これらの適用条件を正確に求め，適用可能なプログラムの範囲を検証することが課題である．

また，本研究では再帰型のマージソートと同じ処理を模倣することに重点を置いた．そのため，適切な部分に処理を挿入することで，先頭再帰以外の再帰の型にも応用できるかもしれない．この点についてはまだ検討中である．逆に，具体的な実行順序や領域の選び方が，元のプログラムと同じでなくても動作結果が変わらないクラスのみを考えることによって，より高速な反復型のプログラムを生成することも可能であると考えられる．個々のプログラム固有のアルゴリズムに立ち入っての再帰除去についても考えて行きたい．

参考文献

[1] A. Pettorossi and M. Proietti : Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, Vol. 28, No. 2, June 1996, pp. 360–414.

[2] Y. A. Liu and S. D. Stoller : From recursion to iteration: what are the optimizations? . *Proceedings of the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Janualy 2000, pp. 73–82.

[3] 二村良彦，大谷啓記：線形再帰プログラムからの再帰除去とその実際的効果．コンピュータソフトウェア，Vol. 15，No. 3，3月，1998，pp. 38–49.

[4] 市川 裕輔，小西善二郎，二村良彦：単一後継関数を持つ相互再帰プログラムに対する再帰除去法の実装．日本ソフトウェア科学会第21回大会論文集，7C-1，9月，2004.

[5] D. A. Bacon, S. L.Graham and O. J. Sharp : Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, vol. 26, No. 4, December 1994, pp. 345–420.

[6] 高須洋平：配列を扱う非線形再帰プログラムの再帰除去について．2004年度 夏の LA シンポジウム，7月，2004，pp. S6-1–S6-4.

A ソースコード

今回，プログラムの実装にはC言語を用いた．以下にそのコードを示す．また，プログラム中の関数 merge は，配列中のある領域を受け取って，その前半部と後半部をマージするための関数である．

merge : マージ関数

```
/* 配列 a 中の p 番目から n 個の要素を二分割し，
   前半と後半をマージする */
void merge(int a[], int b[], int p,int n)
{
  int i, j, k, h;
  h = n / 2;
  i = p;
  j = p+h;
  for(k = p; k < p+n; k++){
    if(j == p+n | ((i<p+h)&(a[i]<=a[j]))
    {
      b[k] = a[i++];
    }else{
      b[k] = a[j++];
    }
  }
  for(k=p; k<p+n; k++)  a[k] = b[k];
}
```

ordinary_msort : 再帰によるマージソート関数

```
/* 配列 a 中の p 番目から n 個の要素に
   マージソートを適用する */
void ordinary_msort
(int a[], int b[], int p, int n)
{
  int h;
  if(n > 1){
    h = n/2;
    ordinary_msort(a, b, p, h);
    ordinary_msort(a, b, p+h, n-h);
    merge(a, b, p, n);
  }
}
```

array_msort : 配列を用いたマージソート関数

```
/* Set 構造体は (int size, int depth) の二項組 */
typedef struct int\_[]set{
  int size;
  int depth;
} Set;

/* 配列 a 中の p 番目から n 個の要素に
   マージソートを適用する */
void array_msort
(int a[], int b[], int p, int n)
```

```
{
  int i, h, m, start, next;

 /* 領域のパラメータを記憶する配列 range */
  Set *range;

  if(n > 1){
    range = (Set *)malloc(sizeof(Set) * n);
    for(i=0; i < n; i++) {
      range[i].depth = 0;
      range[i].size = 0;
    }

    /* 分割部 */

    /* start から range[start].size 個の領域を
       二分割していく */

    start = p;  /* 初期は pから n個の領域 */
    range[start].size = n;
    range[start].depth = 0;

    while(start < p+n){

      /* 領域のサイズが 2 以上であれば分割を行う. */
      if(range[start].size > 1){

        /** ここに処理を入れれば,
            先頭再帰でなくても実装可能か? **/

        m = range[start].size;
        h = m/2;

        /* 分割後の領域のサイズを格納.  */
        range[start].size = h;
        range[start+h].size = m - h;
        /* depth の値を更新 */
        range[start].depth++;
        range[start+h].depth
        = range[start].depth;
      }
      /* サイズ 1 の場合, 処理せずにパス.  */
      else { start++; }
    }

    /* 統合部 */

    /* depth が 0 になるまで繰り返す.  */

    while(range[p].depth > 0){
      /* 最も左の領域を取る */
      start = p;
      /* 隣接した領域を取る */
      next = start + range[start].size;

      /* depth が一致しなければ, 次の範囲を取る. */
      while
      (range[start].depth != range[next].depth)
      {
        start = next;
        next = start + range[start].size;
      }

      /* 二つの領域の合計サイズを計算し,
         merge する.  */
      m = range[start].size + range[next].size;
      merge(a, b, start, m);

      /* 統合した領域のサイズを格納し,
         depth の値を減らす.  */
      range[start].size = m;
      range[start].depth--;
    }
    free(range);
  }
}
```